HYPERBLOCK PERFORMANCE OPTIMIZATIONS FOR ILP PROCESSORS

BY

DAVID ISAAC AUGUST

B.S., Rensselaer Polytechnic Institute, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his guidance and support. Credit is due to the IMPACT group for creating such a useful framework upon which this work was built. I also wish to acknowledge the IMPACT group for graciously offering assistance in many forms. I am especially indebted to Scott Mahlke. His advice, guidance, and assistance were extremely valuable. John Gyllenhaal kindly enhanced the IMPACT profiler at my request. The development and testing that this involved is greatly appreciated. The Office of Naval Research and the University of Illinois provided financial support through fellowships.

I am ever grateful to my family for their love, encouragement, and support. Thank you Dad for being a role model and for sparking my interest in this field as early as 1976. Thank you Mom for, as you frequently remind me, teaching me everything *you* know about computers. Also, thank you for the insight, guidance, and values by which I live. Finally, to Kathy, whose love and strength were vital, and to Murphy, whose loyal companionship during many late nights kept me human, thank you.

TABLE OF CONTENTS

Page

1. INTRODUCTION

Superscalar and very long instruction word (VLIW) processors provide significant performance improvements over scalar processors by simultaneously executing multiple instructions. The effectiveness of these processors depends on the ability of compilers to provide sufficient instruction-level parallelism (ILP) in program code. However, recent studies show that conventional code optimization and scheduling methods cannot provide enough ILP to obtain a sustained speedup of more than two for nonnumeric programs [1],[2],[3]. The high frequency of conditional branch instructions in nonnumeric programs is mostly responsible for these poor results.

Branch instructions impede the ability of the compiler to extract ILP in several ways. Branches impose restrictions on the ability of the compiler to move code. Moving instructions before branches is termed speculation. Speculation is difficult to perform across many branches. Special techniques are usually necessary to handle exceptions that speculated instructions may falsely introduce during execution. In addition, not all instructions are easily speculated. Specifically, speculating stores to memory and branch instructions is particularly problematic. These code motion limitations limit the freedom of the compiler to schedule independent instructions together.

Another way in which branches adversely affect ILP is their long latency. As machines take on longer pipelines, the time between the execute stage and the fetch stage is increased. This extra time translates into large bubbles in the pipeline whenever a branch is encountered. A common technique to reduce the effect of branch latency is branch prediction. Branch prediction eliminates pipeline bubbles by speculatively executing one branch destination path. However, some branches have poor predictability, thus reducing the gains obtained by branch prediction. Branch mispredictions penalize performance by creating many wasted cycles, which amortize away any gains obtained through ILP.

The final reason branches impede ILP so severely is that frequent branches in the instruction stream place an upper limit on the potential ILP. A superscalar or VLIW processor must likely execute multiple branches per cycle to sustain the execution of multiple instructions per cycle. Assuming that an instruction stream contains 25% branches, an 8-issue superscalar processor must have the capability to sustain at least two branches per cycle. If a given 8-issue processor can only execute a single branch each cycle, its maximal performance would be resource limited to four instructions per cycle. Handling multiple branches per cycle requires additional pipeline complexity, as well as designing multiported branch prediction structures such as the branch target buffer (BTB). For this reason, it is likely that most future generation ILP processors will have limited branch handling capabilities. In high issue rate processors, it is less expensive to duplicate arithmetic function units than to predict and execute multiple branches per cycle. Adding extra functional units to compensate for limited branch hardware, if possible, is quite appealing.

Extracting ILP in nonnumeric programs requires that their branch characteristics be improved. One method of achieving this is a technique known as *predicated execution*. Predicated execution refers to the conditional execution of an instruction based on the value of a Boolean source operand, referred to as the predicate [4],[5]. Such architectural support allows the compiler to use an *if-conversion* algorithm to convert undesirable conditional branches into predicate define instructions and instructions along alternative paths of each branch into predicated instructions [6],[7],[8],[9],[10].

Predicated execution facilitates code motion by removing branches and forming structures known as hyperblocks. A hyperblock is a structure formed by combining basic blocks from many paths of execution together. Hyperblocks are optimized and scheduled easily as a unit [9]. The branch misprediction penalty problem can also be alleviated by predicated execution. With predication, many hard to predict branches are eliminated through if-conversion. If the remaining branches are easily predicted, near perfect branch prediction is obtained. Finally, the removal of branches improves the branch handling complexity by reducing the number of branches that must be executed every cycle.

In addition to improving the branch characteristics of nonnumeric programs, predication enables a new class of optimizations, which previously were difficult or even impossible to perform. This thesis describes a set of such optimizations and explores one such optimization, loop peeling, in detail.

1.1 Related Work

Loop peeling, as studied in this thesis, was originally mentioned in [11]. Mahlke's Ph.D. thesis discusses the possibility of performing loop peeling automatically [12]. In fact, performance gains obtained in [12] are due in part to the automatic loop peeling heuristic and optimization developed as part of this work.

The fully resolved predication techniques and other height reduction techniques are studied in [13], and node splitting is briefly mentioned in [11]. Reverse if-conversion, discussed in [14], inspired the idea of partial reverse if-conversion. Discussions that the author had with members of the IMPACT compiler group inspired the ideas of profile independent hyperblock selection and optimization during scheduling.

1.2 Organization of this Work

The following chapter provides an overview of the IMPACT compiler used to study the effectiveness of the techniques presented in this thesis. Chapter 3 provides a detailed discussion of loop peeling and how, together with predication, it is an effective technique to extract ILP from programs. This chapter also evaluates loop peeling on an experimental level and demonstrates its effectiveness in real programs. A preliminary study of a suite of advanced predication compilation techniques is presented in Chapter 4. Finally, Chapter 5 finishes this work with concluding remarks.

2. OVERVIEW OF THE IMPACT COMPILER

All of the compiler techniques necessary to effectively utilize speculative and predicated execution are implemented within the framework of the IMPACT compiler. A block diagram of the IMPACT compiler is presented in Figure 2.1. The compiler is divided into two distinct parts based on the level of intermediate representation (IR) used. The highest level IR, *Pcode*, is a parallel C code representation with loop constructs intact. In Pcode, memory dependence analysis [15],[16], loop-level transformations [17], and memory system optimizations [18],[19] are performed. Additionally, profile-guided code layout and function inline expansion are performed at this level [20],[21],[22].

The lowest level IR in the IMPACT compiler is referred to as *Lcode*, which is a generalized register transfer language similar in structure to most load/store processor assembly instruction sets. Lcode is logically subdivided into two subcomponents, the machine independent IR, Lcode, and the machine specific IR, *Mcode*. The data structures for both the Lcode and Mcode are identical. The difference is that Mcode is broken down such that there is a one-to-one mapping between Mcode instructions and the target

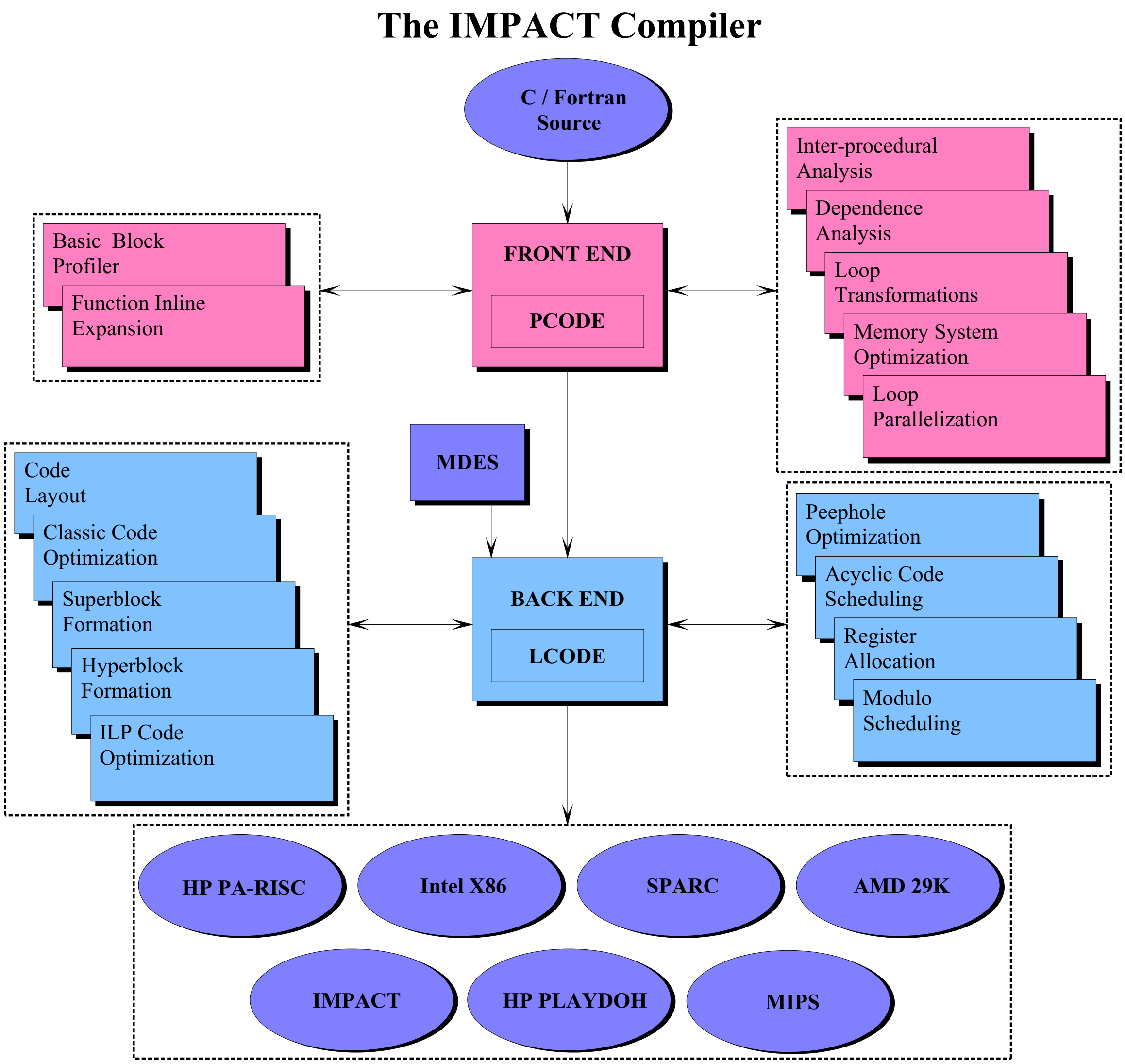


Figure 2.1: The IMPACT compiler.

machine's assembly language instructions. Therefore, to convert Lcode to Mcode, the code generator breaks up Lcode instructions into one or more instructions that directly map to the target architecture. Lcode instructions are broken up for a variety of reasons including limited addressing modes, limited opcode availability, ability to specify a literal operand, and field width of literal operands.

At the Lcode level, all machine independent classic optimizations are applied [23]. These optimizations include constant propagation, forward copy propagation, backward copy propagation, common subexpression elimination, redundant load elimination, redundant store elimination, strength reduction, constant folding, constant combining, operation folding, operation cancelation, code reordering, dead code removal, jump optimization, unreachable code elimination, loop invariant code removal, loop global variable migration, loop induction variable strength reduction, loop induction variable elimination, and loop induction variable reassociation. Additionally, at the Lcode level, interprocedural safety analysis is performed [24], including the identification of safe instructions for speculation and function calls that do not modify memory (side-effect free).

Superblock and hyperblock compilation techniques are performed exclusively at the Lcode level. Superblock formation using execution profile information, superblock classical optimization, and superblock ILP optimization are all supported. When predicated execution support is available in the target architecture, hyperblocks, rather than superblocks, are used as the underlying compilation structure. All superblock optimization techniques have been extended to operate on hyperblocks. In addition, a set of hyperblock-specific optimizations is employed to further exploit predicated execution support. The focus of this thesis is centered around enhancing this set of hyperblock-specific optimizations.

Code generation in the IMPACT compiler is performed at the Lcode level. The two largest components of code generation are the instruction scheduler and register allocator. Scheduling is performed via either acyclic global scheduling [24],[25] or software pipelining using modulo scheduling [26]. For the acyclic global scheduling, code scheduling is applied both before register allocation (prepass scheduling) and after register allocation (postpass scheduling) to generate an efficient schedule. For software pipelining, loops targeted for pipelining are identified at the Pcode level and marked for pipelining. These loops are then scheduled using software pipelining, and the remaining code is scheduled using the acyclic global scheduler. In addition to control speculation, both scheduling techniques are capable of exploiting architectural support for data speculation to achieve more aggressive schedules [16],[27],[28].

Graph coloring-based register allocation is utilized for all target architectures [29]. The register allocator employs execution profile information, if it is available, to make more intelligent decisions. For each target architecture, a set of specially tailored peephole optimizations is performed. These peephole optimizations are designed to remove inefficiencies during Lcode to Mcode conversion, to take advantage of specialized opcodes available in the architecture, and to remove inefficient code inserted by the register allocator.

A detailed machine description database, *Mdes*, for the target architecture is also available to all Lcode compilation modules [30]. The Mdes contains a large set of information to assist with optimization, scheduling, register allocation, and code generation. Information, such as the number and type of available function units, size and width of register files, instruction latencies, instruction input/output constraints, addressing modes, and pipeline constraints, is provided by the Mdes. The Mdes is queried by the optimization phases to make intelligent decisions regarding the applicability of transformations. The scheduler and register allocator rely more heavily on the Mdes to generate efficient and correct code.

Seven architectures are actively supported by the IMPACT compiler. These include the AMD 29K [31], the MIPS R3000 [32], the SPARC [33], the HP PA-RISC, and the Intel X86. The other two supported architectures, IMPACT and HPL Playdoh [34], are experimental ILP architectures. These architectures provide an experimental framework for compiler and architecture research. The IMPACT architecture is a parameterized superscalar processor with an extended version of the HP PA-RISC instruction set. Varying levels of support for speculative execution and predicated execution are available in the IMPACT architecture. For this thesis, all experiments utilize the IMPACT architecture with varying parameters.

3. HYPERBLOCK LOOP PEELING

Creation of a hyperblock consists of two parts: the first is block selection. Block selection is the process of deciding which blocks in a region to include in a hyperblock. Conventional techniques for if-conversion predicate all blocks within a single-loop nest region together [35]. However, for hyperblocks, a subset of these blocks is chosen in order to achieve the best possible performance. Including too few or too many blocks reduces the effectiveness of the formed hyperblock in various ways.

After block selection, hyperblock formation is performed by if-converting the selected blocks together. Before if-conversion, two conditions must be satisfied.

- *Condition 1:* There exist no incoming control flow arcs from blocks not selected to any selected blocks except the entry block.

- *Condition 2:* There exist no loops inside the selected blocks that do not include the entry block.

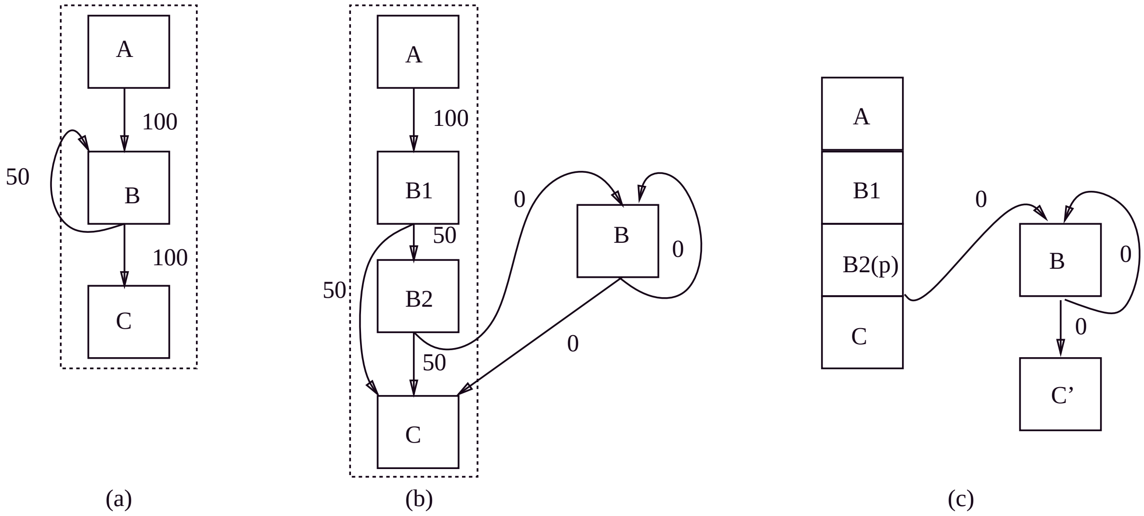


Figure 3.1: Example weighted control flow graph to illustrate hyperblock loop peeling: (a) original graph, (b) graph after loop peeling, and (c) graph after loop peeling, tail duplication, and if-conversion.

Together these conditions insure that the hyperblock has a single entry point and that any instruction in the hyperblock is fetched no more than one time before the hyperblock is exited. Condition 1 is handled routinely by tail duplication. Condition 2 can be satisfied by a transformation known as loop peeling.

Loop peeling is the process of converting the first N iterations of a loop into acyclic code. Figure 3.1 demonstrates this process. Figure 3.1(a) shows the original control flow graph with a dotted line indicating the region chosen by block selection for inclusion into a hyperblock. Block B is a self loop in this region. In order to satisfy Condition 2, this loop is peeled with $N = 2$. To peel a loop, N copies of the loop are made. These copies are exactly the same as the original loop with one exception. The loop backedge destination is changed to point to the next iteration instead of to the header of the loop. Note that the original loop remains untouched, which is necessary for the cases where the loop iterates more than N times. The original copy of the loop is now only executed when the number of iterations exceeds the number of peels. For this reason, the original copy is called the *recovery loop*. Figure 3.1(b) shows the resulting control flow graph after loop peeling.

After loop peeling, the control flow graph in Figure 3.1(b) satisfies Condition 2. Before the hyperblock is formed, tail duplication must be performed to eliminate the control flow arc into block C. Once tail duplication is complete, if-conversion can be performed. The resulting hyperblock is shown in Figure 3.1(c).

While loop peeling itself is simple, knowing when and how much to peel is a complicated problem. The ability to peel makes block selection more difficult. Determining the number of times to peel is crucial to obtaining good performance. Too few peels will result in a hyperblock that seldom completes, while too many peels will result in an increased cycle count for completion of the hyperblock. Resource utilization, dependence height, and the loop's behavior must all be considered. The remainder of this chapter addresses these issues.

3.1 Opportunities for Hyperblock Loop Peeling

Rather than peeling loops for inclusion into a hyperblock, it is possible to perform block selection so that nested loops are not selected. However, this algorithm limits opportunities in hyperblock formation. To understand why block selection should be allowed to include loops for peeling, it is important to study the potential benefits of loop peeling.

The goal of hyperblock formation is to choose a few, very likely straight-line paths and combine them into a single fetch stream. In general, these paths may or may not have loops. Loop peeling allows loops to be converted into straight-line paths. In this way, frequently executed paths are handled in a uniform manner regardless of the loops they may contain.

When a loop is frequently invoked, yet iterated upon no more than a few times per invocation, it may not be possible to enhance the loop's ILP with conventional techniques. This situation is due to the fact that many conventional ILP loop optimizations, such as software pipelining and loop unrolling, rely on loops to iterate many times upon each invocation. These transformations also require that each iteration of the loop is not very sequential since they overlap iterations to increase throughput. However, if the loop is sequential, the iterations cannot be overlapped. Additionally, if the loop iterates infrequently for each invocation, there may not be enough iterations to overlap, let alone amortize any start-up costs involved. Loop peeling is not subject to these constraints. Loop peeling enhances the ILP of these troublesome loops by overlapping their execution with the rest of the path chosen during block selection. Thus very sequential loop iterations are executed simultaneously with other code.

Even when a loop is well-behaved and a good candidate for software pipelining or loop unrolling, loop peeling may have an advantage. Sometimes, the path surrounding

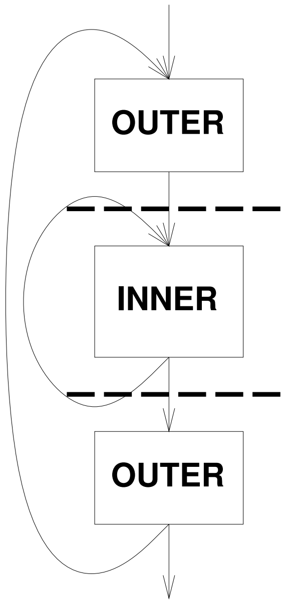


Figure 3.2: Barrier to code motion.

the loop has undesirable characteristics. For example, it may be very sequential or it may not be possible to overlap it with other portions of the region. In this case, loop peeling could provide useful work for overlap with the code of the surrounding region.

Loops on frequently executed paths create artificial barriers to code motion for practical reasons. Figure 3.2 shows a hypothetical loop nest. Since scheduling and optimization across control blocks is difficult, the inner loop creates an artificial barrier to code motion. No longer can the outer loop be treated as a single unit as it has effectively been converted into two separate acyclic code regions. If any or all of these blocks are very sequential, it is necessary to overlap code from each block to create a packed path with high ILP and short cycle count. However, the artificially created barrier to code motion keeps these blocks separate and sparse, resulting in poor performance.

It is clear from this discussion that there are situations where loop peeling is desirable and others where it is not. Determining which loops to peel is the challenge. The decision to peel a loop must be made before the hyperblock is formed. At this point, it is unclear whether peeling will have the desired effect. Depending on the quality of the code optimizations performed after the hyperblock is formed, the choice may be a success or failure. A good heuristic is the best that can be hoped for given this environment. Advanced techniques, such as partial reverse if-conversion and node splitting, which can potentially reduce the dependence on heuristics, are discussed in Chapter 4.

Before the algorithms used in the IMPACT hyperblock formation module are presented in Section 3.4, rough guidelines for finding good candidates for peeling will be presented.

Low iteration count loops are generally good candidates for peeling. Consider a loop that iterates three or fewer times per invocation. Since there are only a few iterations, the number of peels is minimal. These peels are likely to be simultaneously executable with other code in the selected traces. Additionally, since the number of iterations is usually less than the number of peels, the program is unlikely to exit the hyperblock before completion. As mentioned earlier, unrolling and software pipelining cannot enhance this loop's performance because it does not have a high iteration count.

A loop that has a low iteration count on some invocations and a very high iteration count on others is also a good candidate for loop peeling. The low iteration count invocations have the same advantages discussed earlier. The high iteration count invocations will always exit to the recovery loop. Whenever the recovery loop is executed, it is likely that the loop will still need to iterate for many iterations. For this reason, high iteration count loop transformations are still effective. The high iteration count recovery loop can be optimized while safely ignoring the low iteration count cases. Any start-up penalty will be amortized in all cases. In effect, the loop is versioned. The peeled loop is the version for low iteration counts, whereas the recovery loop is the version for high iteration counts.

Even when the loop does not have a low iteration count component or the behavior of the loop is simply unknown, it may still be wise to peel it. If the code surrounding the loop is sparse, peeling off a few iterations to utilize resources will increase ILP. In this case, determining the number of times to peel is more difficult. The dependence height and resource usage of the resulting peels must be matched carefully with the characteristics of the surrounding code. The process is made more difficult since the result of further transformations and optimizations must be considered.

In all cases, the number of times to peel a loop is critical. Too few iterations result in the recovery loop being entered too often, resulting in an ineffective hyperblock. Too many peels result in many useless instructions wasting fetch resources or in an unnecessarily increased dependence height.

```
        /* OUTER LOOP */
        for (p = CC−>data, last = p + CC−>count * CC−>wsize;
LG:          p < last;
             p += CC−>wsize)
           {
LA:            if (( p[0] & (0x2000) ))
                 {
LB:                  register int i_ = ( p[0] & 0x03FF );
                     /* INNER LOOP */
                     do {
LC:                          if ( p[i_] & ~r[i_])  break;
LD:                      } while(−−i_ > 0);
LE:                  if ( i_ != 0 )
                         goto false1;
                     continue;
                     false1:
LF:                  CC−>active_count−−,  ( p[0] &= ~(0x2000));
                 }
           }
```

Figure 3.3: Source code of a loop nest in *elim_lowering* from *008.espresso*.

3.2 An Example Code Segment

In order to gain a better understanding of how peeling works, a code example is presented. Figure 3.3 shows the C source code to an important loop nest in *elim_lowering* from the SPEC-92 benchmark *008.espresso*. The inner loop in this code only iterates an average of 2.6 times per invocation. However, this inner loop accounts for a significant portion of the entire benchmark's running time. In addition to being invoked quite frequently, the outer loop iterates an average of 626 times per invocation causing the inner loop to be invoked frequently. The outer loop would have been ideal for loop unrolling or software pipelining if not for the inner loop.

```
1   LA: r98 = MEM[R1]
2       r99 = r98 & 8192
3       branch (r99 == 0), LG
4   LB: r11 = r98 & 1023
5       r115 = r11 << 2
6       r124 = r115 + r1
7       r125 = r115 + r2
8   LC: r56 = MEM[r124]
9       r124 = r124 - 4
10      r58 = MEM[r125]
11      r125 = r125 - 4
12      r59 = -1 XOR r58
13      r60 = r56 & r59
14      branch (r60 != 0), LE
15  LD: r11 = r11 -1
16      branch (r11 > 0), LC
17  LE: branch (r11 == 0), LG
18  LF: r137 = r137 - 1
19      r64 = r98 & -8193
20      MEM[r1] = r64
21  LG: r1 = r1 + r101
22      branch (r1 < r3), LA
```

Figure 3.4: Intermediate representation of *elim_lowering* loop before peeling.

Figure 3.4 is the IMPACT compiler generated intermediate representation of the loop nest. The inner loop is highlighted showing that the inner loop consists of nine operations and has a dependence height of five cycles. Without further transformation, this inner loop would have an IPC of less than two and would be unable to fully utilize wider-issue processors. In addition to the serial nature of the inner loop, the inner loop contains two hard to predict branches. These branches, instructions 14 and 16, are hard to predict because of the low iteration count behavior of the loop. These branches are not strongly biased in the way a frequently taken backedge would be, such as instruction 22, for example.

The outer loop is also quite serial with a total of 13 instructions; its dependence height is seven cycles. Without loop peeling, this code segment would have been optimized and scheduled in three parts. In Figure 3.5, the three portions are shown. Basic blocks A and B make up the first control block corresponding to the top portion of the outer loop. Basic blocks C and D form the inner loop. The bottom portion of the outer loop consists of basic blocks E, F, and G.

The characteristics of this loop nest indicate that the inner loop is a good candidate for peeling. With an average iteration count of 2.6, the inner loop could be peeled three times to capture 90% of all of the loop's invocations. Its small size means that peeling the loop three times would not oversaturate available resources in an 8-issue machine.

Figures 3.6 and 3.7 show the loop nest after loop peeling. While the actual code is peeled four times, the figures show a version of the loop nest peeled twice for demonstration purposes. The first peel, instructions 8 through 16, must always execute. The predicate P130 corresponds to the condition of the branch instruction 3 in Figure 3.4. When P130 is TRUE, the inner loop is not bypassed, and the first iteration of the inner loop will execute. For this reason, the first iteration is predicated on P130. P134 corresponds to the branch condition in instruction 14 in Figure 3.4. The second peeled iteration is predicated on P135 and P130. These predicates are analogous to P130 and P134 with the added backedge condition. If the two peeled loop iterations are insufficient to satisfy the number of iterations, control is directed to the recovery loop by branch instruction 25.

After peeling, the control flow graph has been converted to a single loop with one backedge and one exit edge. Certainly, this resulting loop is more conducive to further

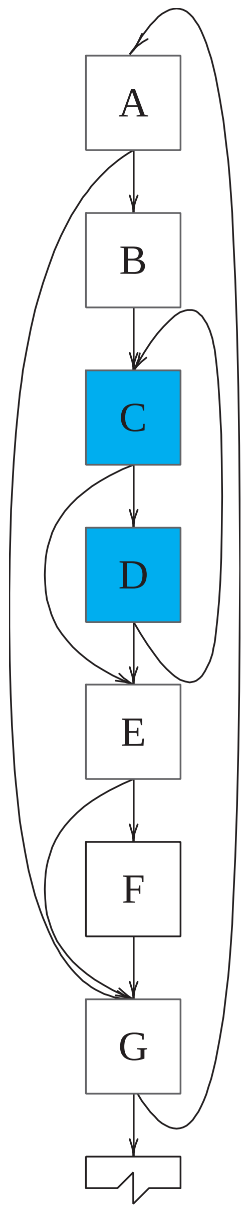


Figure 3.5: Control flow graph of *elim_lowering* loop before peeling.

LA:	r98 = MEM[R1]	
	r99 = r98 & 8192	
	(p130) = (r99 == 0)	
	r11 = r98 & 1023	(p130)
	r115 = r11 << 2	(p130)
	r124 = r124 - 4	(p130)
	r125 = r125 - 4	(p130)
	r56 = MEM[r115 + r1]	(p130)
	r124 = r124 - r115	(p130)
	r58 = MEM[r115 + r2]	(p130)
	r125 = r125 + r115	(p130)
	r59 = -1 XOR r58	(p130)
	r60 = r56 & r59	(p130)
	(p134) = (r60 == 0)	(p130)
	(p135) = (r11 > 1)	(p134)
	r11 = r11 -1	(p134)
	r56 = MEM[r124]	(p135)
	r124 = r124 - 4	(p135)
	r58 = MEM[r125]	(p135)
	r125 = r125 -4	(p135)
	r59 = -1 XOR r58	(p135)
	r60 = r56 & r59	(p135)
	(p136) = (r60 == 0)	(p135)
	r11 = r11 -1	(p136)
	branch (r11 > 0), EXTRA	(p136)
	(p133) = (r11 != 0)	(p130)
	r137 = r137 - 1	(p133)
	r64 = r98 & -8193	(p133)
	MEM[r1] = r64	(p133)
	r1 = r1 + r101	
	branch (r1 < r3), LA	

Figure 3.6: Intermediate representation of *elim_lowering* loop after peeling.

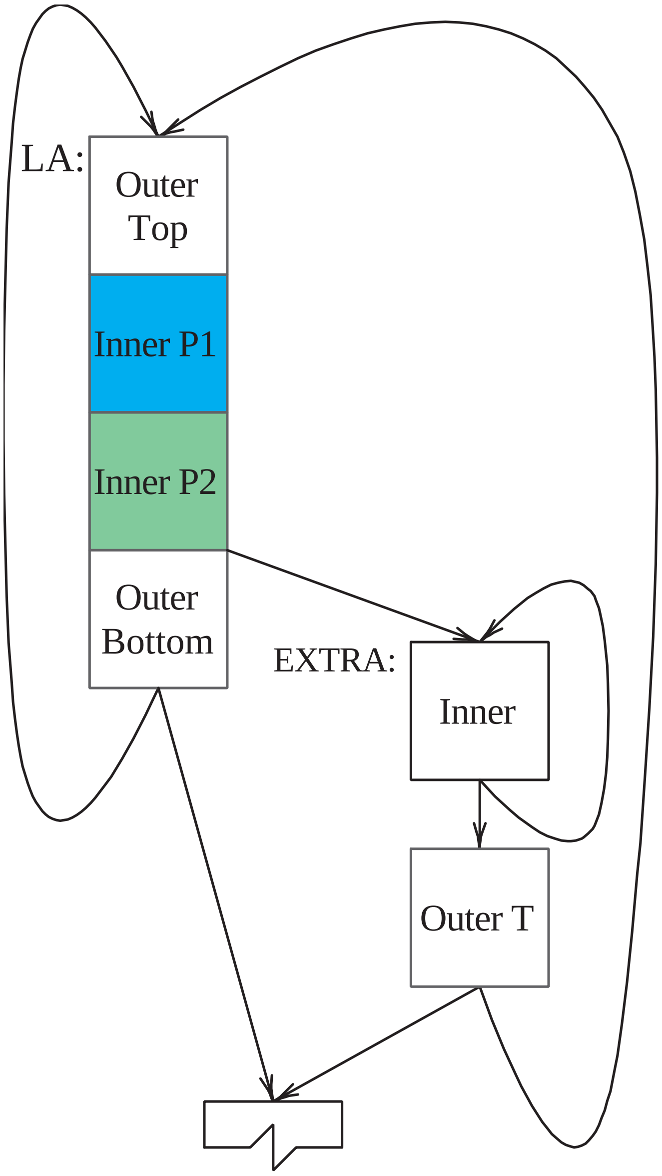


Figure 3.7: Control flow graph of *elim_lowering* loop after peeling.

0	top					
1						
2	top	top				
3	top	top				
4	peel 1	top	peel 1	top		
5	peel 2	peel 2	peel 3	peel 3	peel 4	peel 4
6	peel 1					
7	peel 1	peel 2	peel 3	peel 4		
8	peel 1	peel 2	peel 3	peel 4		
9	peel 1	peel 1				
10	peel 2					
11	peel 2	peel 2				
12	peel 3					
13	peel 3	peel 3				
14	peel 4					
15	peel 4					
16	bottom	extra				
17	bottom	bottom	bottom			
18	backedge					

Figure 3.8: Schedule of *elim_lowering* loop after peeling four times.

optimization. However, even without further optimization, the resulting loop has more desirable characteristics. As is shown in Figure 3.8, many peeled loop iterations are executed concurrently. The first iteration is dependence height limited and takes six cycles. However, each additional iteration overlaps most of its previous iteration, taking only two more cycles. Thus, four iterations take up 12 cycles, only double the cycle count of a single iteration. Additionally, the top and bottom parts of the outer loop segment do overlap with the peels. Unfortunately, in this example the outer loop overlaps a relatively small amount in comparison with other loops. Even with the peeling, the loop is still dependence height limited. Further optimizations such as unrolling will be applied to alleviate this problem.

The final optimized code segment's cycle count has dropped from 6.70 million cycles to 2.46 million cycles due to loop peeling. Loop peeling has increased the instructions per cycle (IPC) by the overlapping of parts of the inner loop. However, the most interesting change is observed when branch prediction is modeled. The number of mispredictions is reduced from 182,000 to only 16,000. This reduction in mispredictions is the effect of removing the hard to predict inner loop branches. In a machine with a large branch misprediction penalty, this reduction in mispredictions can have an even more profound effect on the cycle count.

3.3 Loop Iteration Histogram Profiling

An important characteristic to consider when deciding which loops to peel and how many times to peel them is the iteration histogram. The iteration histogram is a count of the number of times a loop is iterated each time it is invoked. Figure 3.9 shows an Lcode loop header basic block with loop iteration histogram information. This basic block corresponds to the header of the inner loop in the *008.espresso* code segment discussed previously. The *iteration_header* attribute indicates that each time the loop was entered it had only four different iteration counts. It also indicates that this loop was profiled only once. The four *iter_N* attributes indicate the number of times the loop iterated N times when it was invoked. In this case the loop had one iteration before exiting in 3400 of its invocations. It also had two iterations in 57,481 of its invocations, three iterations in 52,233 of its invocations, and four iterations in 13,849 of its invocations.

```
(cb 11 330457.000000 [(flow 1 13 125018.000000)(flow 0 12 205439.000000)]
      <(iteration_header (i 4)(i 1))
       (iter_1 (f2 3400)(f2 3400))
       (iter_2 (f2 57481)(f2 57481))
       (iter_3 (f2 52233)(f2 52233))
       (iter_4 (f2 13849)(f2 13849))>)
  (op 77 ld_i [(r 56 i)] [(r 124 i)(i 0)])
  (op 177 add [(r 124 i)] [(r 124 i)(i -4)])
  (op 79 ld_i [(r 58 i)] [(r 125 i)(i 0)])
  (op 179 add [(r 125 i)] [(r 125 i)(i -4)])
  (op 80 xor [(r 59 i)] [(i -1)(r 58 i)])
  (op 81 and [(r 60 i)] [(r 56 i)(r 59 i)])
  (op 82 bne [] [(r 60 i)(i 0)(cb 13)])
```

Figure 3.9: Lcode example of loop iteration count histogram profiling.

These attribute values are obtained by dynamic profiling. Profiling is performed by running the benchmark with probes in the code to report its behavior. Each *iter_N* attribute contains at least two numbers. The first number is the average number of invocations among all profile runs. The subsequent numbers are the per run count. In this case, only one profile run was taken; hence, there are only two numbers, both of which are equal.

Once collected, this loop iteration profile information is used to decide when and how much to peel a loop. One concern over utilizing profile information is that it may not accurately reflect the behavior of the program with different input sets. If the profile information used in loop peeling decisions does not match the behavior of the program, performance suffers. To quantify this effect and justify the use of profiling information in loop peeling decisions, the effect of different input sets will be studied as part of future research. It is suspected that, while the actual iteration count may vary by a small amount, infrequently iterated loops will remain infrequently iterated for most input sets. The next section discusses the algorithms used in the loop peeling decisions. These algorithms assume that profile information is a good estimate for most input sets.

3.4 Loop Peeling Selection Heuristic

With loop peeling, knowing exactly when and how much to peel is impossible since optimizations and scheduling performed after peeling can radically change the characteristics of the loop. For this reason, the peeling decision must be a heuristic.

One way to decide when to peel is to figure out how much peeling would make sense for a particular loop. If the number of peels is reasonable, then peeling that loop may be a wise decision. However, if the number of peels is zero or is very large, then peeling could be detrimental. Deciding how much to peel involves several issues. Peeling too few times would mean that the recovery loop would be invoked too often, making the original hyperblock ineffective. Peeling too many times may introduce many useless instructions into the hyperblock. These useless instructions saturate the fetch resources and increase the cycle count of the hyperblock. The currently implemented heuristic shown in Figure 3.10 attempts to balance these concerns to find the best number of peels. It also indirectly determines if a loop is a candidate for peeling. If a good number of peels cannot be found, the loop is not peeled.

```
find_num_peel(loop)
{
    if(contains_jsr(loop))
    {
        return 0;
    }

    total_invocations = compute_total_invocations(loop);
    total_peelable_invocations = compute_total_peelable_invocations(loop);
    iteration_size = compute_iteration_size(loop);

    overall_coverage = 0;
    peelable_coverage = 0;

    for(cur_num_peel = 0;
        cur_num_peel < CONSIDER_INFINITY_ITERATIONS;
        cur_num_peel = cur_num_peel + 1)
    {
        overall_coverage = overall_coverage +
            (iter_count[cur_num_peel] / total_invocations);
        incremental_peelable_coverage =
            iter_count[cur_num_peel] / tot_peelable_iter;
        peelable_coverage = peelable_coverage + incremental_peelable_coverage;

        if (overall_coverage < MIN_OVERALL_COVERAGE)
            continue;
        if (incremental_peelable_coverage < MIN_PEELABLE_INCREMENTAL_COVERAGE)
            continue;
        if (peelable_coverage < MIN_PEELABLE_COVERAGE)
            continue;
        if ((cur_num_peel * iteration_size) > MAX_OPS_IN_PEELED_LOOP)
            break;

        return cur_num_peel;
    }

    return 0;
}
```

Figure 3.10: Heuristic to compute number of times to peel a loop.

The current heuristic will return 0, meaning that the loop should not be peeled if it contains a jsr. Jsr's limit code motion in hyperblocks. For this reason, it is usually not desirable to create multiple calls to functions in a hyperblock by peeling the loop. If the loop does not contain a jsr, it still needs to be further considered.

The heuristic uses five tunable parameters in order to find the best number of times to peel. The logic behind each of these parameters is discussed in turn. The value used in the IMPACT compiler for each of these parameters was determined by testing the code on the SPEC-92 benchmark suite.

The first parameter, *CONSIDER_INFINITY_ITERATIONS*, is the upper bound on the number of peels. In effect, this parameter chooses a threshold at which point a high iteration count loop transformation, such as software pipelining or loop unrolling, is a better choice. The value of this parameter is normally around six or eight. This parameter is critical since the heuristic works by considering each number of peels starting at zero up to *CONSIDER_INFINITY_ITERATIONS*, until the first case is found that satisfies all of the other parameters.

Another parameter is *MAX_OPS_IN_PEELED_LOOP*. This parameter is the maximum number of operations the peeled loop code can have. The number of operations in a peeled loop code segment is considered to be the number of instructions in a loop multiplied by the number of peels. If this number exceeds *MAX_OPS_IN_PEELED_LOOP*, the loop is considered to be a bad candidate for peeling, and the loop is not peeled.

In actuality, a large number of operations in a peeled loop code segment are not necessarily detrimental. However, the number of operations is used as a rough estimate of the dependence height and resource usage a peel may consume. Future implementations of the loop peeler will likely directly consider dependence height. In addition to being more representative of the actual code characteristics, considering dependence height would enable the use of loop peeling in a sparse outer loop case. The value of *MAX_OPS_IN_PEELED_LOOP* is usually around 36.

The *MIN_OVERALL_COVERAGE* is the minimum number of invocations by percentage that a peeled loop needs to execute without needing the recovery code. The coverage a peeled loop has is computed by adding all of the invocation counts for each iteration count from zero to the number of peels currently under consideration and dividing by the total number of invocations of the loop. If this number is less than *MIN_OVERALL_COVERAGE*, the loop is not peeled, and a larger peel count is then considered. This parameter is usually set to 0.75, which means that 75% of all invocations need to be covered.

MIN_PEELABLE_COVERAGE and *MIN_PEELABLE_INCREMENTAL_COVERAGE* relate the invocation count coverage to the total peelable invocations. The total peelable invocation is the number of invocations with iteration counts less than *CONSIDER_INFINITY_ITERATIONS*. *MIN_PEELABLE_COVERAGE* is the minimum ratio of coverage that would be provided by the peeled loop code using the currently considered peel count and the total peelable invocations. It is usually set to 0.85 or 85%.

MIN_PEELABLE_INCREMENTAL_COVERAGE is the minimum amount of extra coverage provided solely by the currently considered peel count. It is usually set to 0.1 or 10%.

As mentioned earlier, the current heuristic does not consider the dependence height of the loop directly. It uses the operation count as a rough measure. Using the dependence height directly together with the operation count, the scheduling constraints on the resulting code could be estimated accurately. In addition to improved accuracy, using dependence directly may enable peeling to be used in more situations. As was discussed in Section 3.1, it is desirable to peel a loop regardless of its iteration characteristics if the code surrounding the loop is sparse. Future versions of the loop peeling heuristic could compute the dependence height of the surrounding code, as well as the loop itself, to find more cases where peeling is appropriate.

3.5 Peeled Loop Optimizations

Classical loop optimizations are designed to process loop structures. When a loop is peeled, it looses its loop structure and is treated like acyclic code. In addition to acyclic code optimizations, a peeled loop should be subject to optimizations that would have been applied to the original loop. To get loop optimizations to operate on the peeled iterations, these optimizations must be modified as is done in the case of unrolled loops.

Fortunately, not all loop transformations need reimplementation. Many have straight-line code equivalents that have the same effect. For example, invariant code removal in loops has the same effect as common subexpression elimination in acyclic code. Two important optimizations, accumulator expansion and induction variable elimination, have no acyclic equivalents in the IMPACT compiler after the loop peeling phase. These straight-line equivalent optimizations have not been implemented because natural code rarely has opportunities for them. However, loop peeling creates many such opportunities as a side effect. The next two sections will discuss these optimizations in turn.

3.5.1 Accumulator expansion

An accumulator is a variable that is repeatedly updated during the execution of a loop. Consider the code in Figure 3.11. The variable *sum* is an accumulator because its final value is the sum of all calls to the function *result*. Unfortunately, when an accumulator exists in a peeled or unrolled loop, the dependence height of the loop may be unnecessarily extended. Consider the sample loop after peeling, but before if-conversion, as shown in Figure 3.12. One reason why this peeled code segment is very serial is that each update of *sum* cannot proceed until the previous update is performed. Accumulator expansion is an optimization technique that is applied in situations such as these to reduce the dependence height.

Figure 3.13 shows the same peeled loop after accumulator expansion. The variable *sum* is replaced by three variables, one for each peeled iteration. Once this is done, no dependences exist between any of the accumulation statements related to the accumulation variable. Note that extra instructions need to be added to accomplish this task. Each

```
for(indx = 0; indx < max; indx++)
    {
        sum = sum + result(indx);
    }
```

Figure 3.11: Loop with accumulator variable.

```
    indx = 0;
    sum = sum + result(indx);
    indx = indx + 1;
    if(indx >= max) goto exit;
    sum = sum + result(indx);
    indx = indx + 1;
    if(indx >= max) goto exit;
    sum = sum + result(indx + 2);
    indx = indx + 1;
    if(indx < max) goto extra_iteration_loop;
```

Figure 3.12: Loop with accumulator variable after peeling.

```
    sum1 = 0;
    sum2 = 0;
    sum3 = 0;
    indx = 0;
    sum1 = sum1 + result(indx);
    indx = indx + 1;
    if(indx >= max) goto exit;
    sum2 = sum2 + result(indx);
    indx = indx + 1;
    if(indx >= max) goto exit;
    sum3 = sum3 + result(indx);
    indx = indx + 1;
    if(indx < max) goto extra_iteration_loop;
exit:
    sum = sum1 + sum2 + sum3;
```

Figure 3.13: Loop with accumulator variable after accumulator expansion.

of the extra accumulators must be initialized to zero. Additionally, the *sum*, which still must be computed, uses an extra statement to combine the partial sums. In optimizing this code segment, accumulator expansion is not sufficient by itself. There still remain other properties that artificially increase the dependence height of the peeled loop. These properties are addressed later.

It is extremely difficult to find all cases where accumulator expansion can be applied. However, a conservative search algorithm can extract the most important cases. Figure 3.14 shows the algorithm used in the IMPACT compiler. A search for instructions meeting certain criteria is applied to each peeled loop in the program. Accumulator registers are found by looking at each definition of all variables in the peeled loop. If all of

```
FOR EACH peeled loop:
    FOR EACH register, rX, where all definitions are in any of the forms,
    { rX = rX + Y; rX = rX − Y; rX = rX ∗ Y;
    rX = rX/Y; rX = Y + rX; rX = Y ∗ rX }:
        Add rX to the list of accumulators.


    FOR EACH rX in the list of accumulators:
        IF rX is defined by only one instruction THEN
            remove rX from the list of accumulators.
        IF rX is used by operations of different accumulator forms THEN
            remove rX from the list of accumulators.

  Accumulator Expansion Consideration:
    FOR EACH rX in the list of accumulators:
        IF rX is used in an instruction which is not of accumulator form THEN

        Goto Induction Variable Elimination Consideration.
        IF Y in each instruction is the same and is a numeric constant
        AND each instruction is either an add or subtract THEN
            Goto Induction Variable Elimination Consideration.
        Perform accumulator expansion.
        Continue with next peeled loop.

  Induction Variable Elimination Consideration:
    FOR EACH rX in the list of accumulators:
        IF Y in each accumulator instruction is the same and is a numeric constant
        AND each instruction is either an add or subtract THEN
            Continue with next peeled loop.
        IF an accumulator instruction is conditionally executed within each iteration THEN
            Continue with next peeled loop.
        Perform induction variable elimination.
```

Figure 3.14: Algorithm for applying accumulator expansion and induction variable elimination.

the definitions of a variable are of the forms listed in Figure 3.14, the variable is added to the list of accumulators. At this point, the list of accumulators contains all potential accumulator variables. However, not all of the accumulators in this list will match the additional criteria needed for correct application of accumulator expansion.

Each accumulator in the list of accumulators must meet the following criteria. If an accumulator is defined by only one instruction, then no optimization is possible. Accumulator expansion applied to a single accumulation instruction has no effect. For this reason, all accumulator variables defined by only one instruction are removed from the list of accumulators. Another criteria for accumulator expansion is that all of the definitions of the accumulator variable are performed by the same operation type because accumulator expansion is only applicable to one operator. For example, it is unclear how accumulator expansion could be applied to an accumulator that had two definitions of the forms $rX = rX + Y$ and $rX = rX/Z$.

As will be discussed further in the next section, accumulators can be optimized in another way known as induction variable elimination. Induction variable elimination optimizes some accumulators more efficiently than accumulator expansion. To select the optimization to which each accumulator is subjected, more criteria are applied. If the accumulator is *used* in an instruction that does not possess one of the original forms shown in the figure, then accumulator expansion cannot be applied safely. However, induction variable elimination does not suffer from this restriction, so it is considered for application. Additionally, if the accumulator instructions are addition or subtraction operations and all have the same numeric constant, then induction variable elimination should be used first since it can generate more efficient code than the more general accumulator variable expansion optimization. If these two criteria do not redirect consideration for optimization to induction variable elimination, then accumulator expansion is applied.

3.5.2 Induction variable elimination

Before the algorithm used to determine whether or not to apply induction variable elimination is considered, an example is presented to illustrate what induction variable elimination does. Figure 3.15 shows the example presented in the previous section after accumulator variable expansion and further optimization. Notice that even though accumulator expansion removed the output dependences among the accumulation instructions, it did not remove the dependences among the induction variable *indx*. Each increment of *indx* is dependent on the previous increment instruction. However, this dependence chain is breakable. At each increment instruction, the result of the computation can be computed simultaneously when it is realized that each instruction is incremented by the same numeric constant. Figure 3.16 shows the result of induction variable elimination on the code. Uses of *indx* are replaced by a new variable that contains the new calculated value. Notice that all uses of *indx* are replaced by different variables depending on their relative position to the original increment instructions. For example, all instructions after the first, but before the second, increment instruction have

```
    indx = 0;
    sum1 = result(indx);
    indx = indx + 1;
    if(indx >= max) goto exit;
    sum2 = result(indx);
    indx = indx + 1;
    if(indx >= max) goto exit;
    sum3 = result(indx);
    indx = indx + 1;
    if(indx < max) goto extra_iteration_loop;
exit:
    sum = sum1 + sum2 + sum3;
```

Figure 3.15: Loop with accumulator variable after accumulator expansion and constant propagation.

```
indx0 = 0;
indx1 = indx0 + 1;
indx2 = indx0 + 2;
indx3 = indx0 + 3;
    sum1 = result(indx0);
    if(indx1 >= max) goto exit;
    sum2 = result(indx1);
    if(indx2 >= max) goto exit;
    sum3 = result(indx2);
    if(indx3 < max) goto extra_iteration_loop;
exit:
    sum = sum1 + sum2 + sum3;
```

Figure 3.16: Loop with induction variable elimination.

indx replaced with *indx1*. Similarly, all instructions after the second, but before the third, increment instruction had *indx* replaced with *indx2*.

Induction variable elimination takes advantage of the fact that for each peeled loop iteration, the number of iterations preceding it is known. If an induction calculation is performed in every iteration, this fact is used to calculate the induction variable value for each peeled iteration in advance. The most important application is to eliminate dependences between *address increment, load* chains because many important loops access arrays based on an induction variable. If the address of each array access is computed early, then all of the loads can be moved earlier. Since most loads start long dependence chains, moving the loads early can be very effective in removing much of the dependence height in a peeled loop body.

Before applying induction variable elimination, the code must meet a few criteria. These criteria are shown at the bottom of Figure 3.14. Induction variable elimination as currently implemented only supports addition and subtraction with a uniform constant. A test in the algorithm verifies that this condition is met by checking the form of the induction variable instructions being considered. If a peeled loop meets this criteria, one final test is performed. If an induction variable is conditionally incremented in each iteration, it cannot be safely optimized because the induction variable elimination optimization needs to compute the final expression for each induction variable at compile time. Since the conditional execution of any induction variable instruction may make the expression uncomputable, the optimization is not performed.

```
    sum1 = result(0);
    if(1 >= max) goto exit;
    sum2 = result(1);
    if(2 >= max) goto exit;
    sum3 = result(2);
    if(3 < max) goto extra_iteration_loop;
exit:
    sum = sum1 + sum2 + sum3;
```

Figure 3.17: Loop with induction variable elimination.

Figure 3.17 shows the example after further classical optimizations are performed. Note that the only dependences in the code before *exit* are control dependences. These control dependences are eliminated later by if-conversion.

3.6 Advanced Hyperblock Loop Peeling

While loop peeling as described is very effective in dealing with some kinds of loops, two techniques presented in this section enhance its effectiveness even further. The first technique, called hyperblock reentry, is a method to reduce the code expansion caused by loop peeling. The second technique, nested loop peeling, allows loop peeling to be applied to deeply nested loops.

As was mentioned in the beginning of this chapter, loop peeling usually requires tail duplication. Figure 3.18(a) shows a loop nest before loop peeling. Figure 3.18(b) shows the peeled loop nest with the tail duplicated portion of the outer loop. This tail duplicated segment of code may make the resulting code less effective. Tail duplication

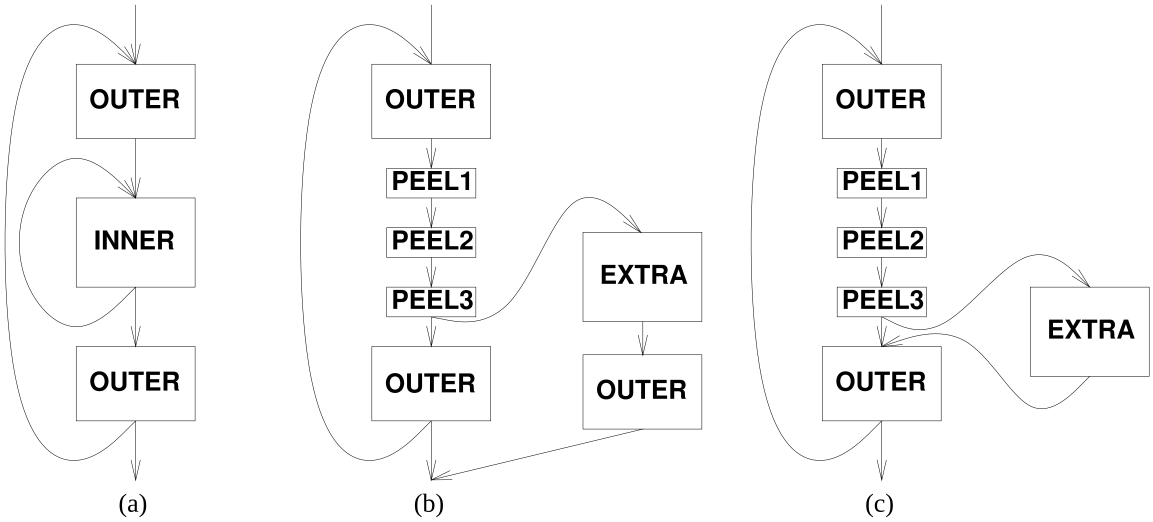


Figure 3.18: Control flow graph illustrating hyperblock reentry: (a) original graph, (b) graph after loop peeling, and (c) graph after loop peeling and hyperblock reentry.

causes extra code expansion, which can result in poor instruction cache performance. Also, tail duplicated code is generally less optimal than on-trace code as less overlapping is possible with other code.

Hyperblock reentry controls the problems caused by tail duplication. Hyperblock reentry, as shown in Figure 3.18(c), eliminated the duplicated portion of the outer loop by jumping back into the hyperblock. Thus hyperblock reentry reduces the code expansion due to tail duplication by eliminating the need for it. It also provides an entry back into the already optimized hyperblock. By jumping back into an already optimized hyperblock, compile time is not wasted optimizing the infrequently executed tail duplicated code segment.

While hyperblock reentry improves the situation for loops that can already be peeled, another technique, known as nested loop peeling, actually allows loop peeling to be used in more situations. A real example in which nested loop peeling is very effective is shown in Figure 3.19. This loop nest contains three loops. The outermost loop with header block 17 is considered to be the outer loop and the header of the formed hyperblock. Block 18 is the header of the middle loop, and it is peeled twice. However, in order for a loop to be peeled, it must be an innermost loop. To transform the middle loop into an innermost loop, the innermost loop, blocks 19 and 20, is first peeled three times. Then, the new inner loop is peeled twice. The final peeled hyperblock is shown at the right of the figure, complete with the associated recovery loop.

Notice that to reduce the code expansion, a single copy of the recovery loop nest is necessary. Three exits from the hyperblock each jump to the correct portion of the recovery loop nest. In this case, the tail duplicated code consists of blocks 29 and 23.

3.7 Experimental Evaluation

Figure 3.20 shows the speedup obtained through the use of loop peeling for five benchmarks. Of these benchmarks only *yacc* and *008.espresso* showed more than a 5% performance improvement. This result is due to the fact that the performance improvement obtained with loop peeling is highly dependent on the structure of the frequently iterated loops. In *008.espresso*, for example, the most time-consuming loop is also a good candidate for peeling. However, in other benchmarks, most of the time is spent in loops that are not good candidates for loop peeling. These loops are poor candidates because they may not have a nested loop structure visible to the compiler. Region-based

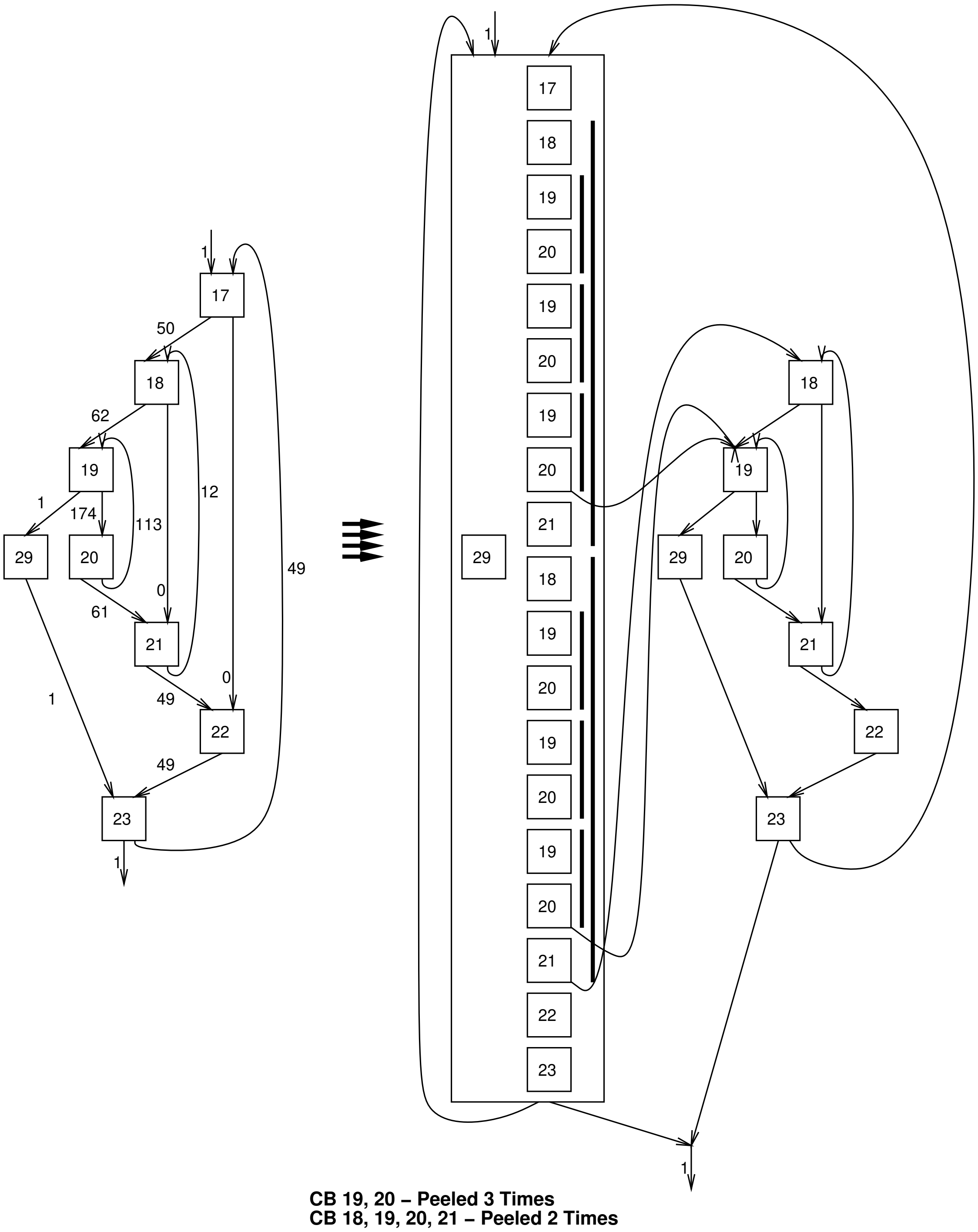


Figure 3.19: A nested loop peeling example from the function *siev*.

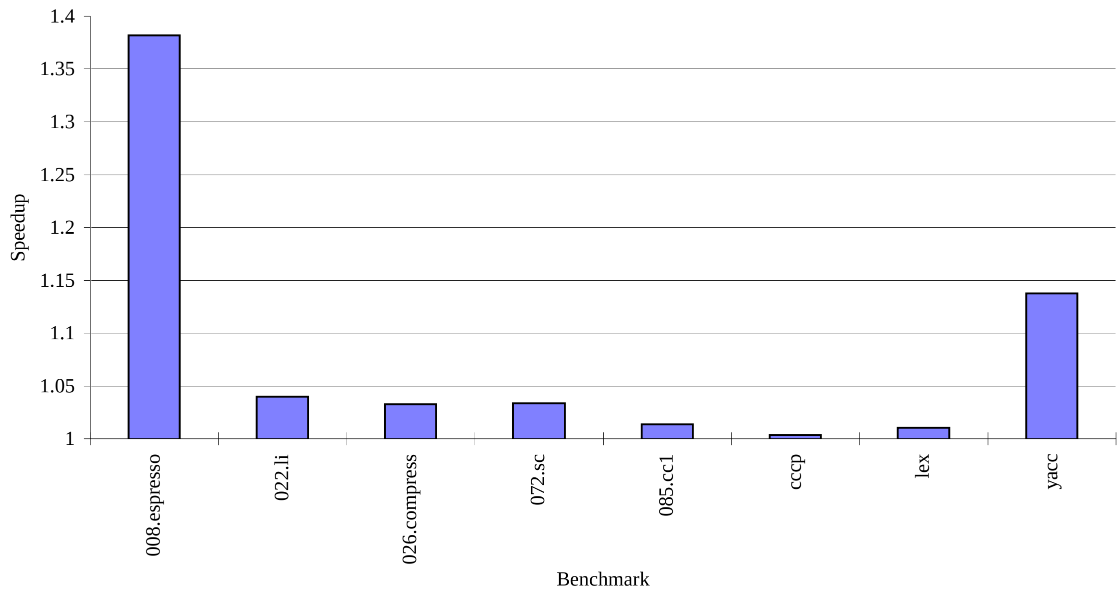


Figure 3.20: Speedup obtained with the use of loop peeling.

compilation may expose more opportunities for peeling as it can expose more of these loop nests [36].

The speedup numbers obtained in Figure 3.20 were generated by compiling multiple times with different sets of parameters for loop peeling selection. The best parameter setting for each function was then selected to compose the final result. The difference between this number and the best parameter setting for an entire benchmark is a good measure for how well the selection heuristic models the resulting code. For *008.espresso*, the best per function cycle count was 101 million cycles, but the best per benchmark cycle count was 110 million cycles. Since *008.espresso* without peeling is 140 million cycles, the heuristic lost 25% of the performance due to inaccuracy. In fact, if a per loop nest number were obtained, a more optimal cycle count would have been computed,

CONSIDER_INFINITY_ITERATIONS	6
MAX_OPS_IN_PEELED_LOOP	36
MIN_OVERALL_COVERAGE	0.60
MIN_PEELABLE_COVERAGE	0.80
MIN_PEELABLE_INCREMENTAL_COVERAGE	0.20

Figure 3.21: Parameter settings yielding the best overall performance.

further illustrating performance loss due to the heuristic. Clearly, there is a need for a more accurate peeling selection heuristic. The heuristic discussed earlier, which takes into account dependence height, would be able to recover some of this lost performance. In no case did the peeling heuristic take advantage of loop versioning and the sparse outer loop cases discussed in Section 3.1. A more robust peeling selection heuristic that took these cases into account would likely realize larger performance gains.

Since it is difficult to study the characteristics of benchmarks with each function using different parameters, a single set of parameters will be used for all benchmarks. The parameter settings used to get the best overall performance are shown in Figure 3.21.

One criticism of loop peeling is that it may cause significant code growth. This code growth originates from the multiple copies of peeled loops generated by peeling. However, measurements made on the amount of code growth show that it is very small. Figure 3.22 shows the relative code size before and after peeling. For all of the benchmarks, program size did not increase by more than 6%. In fact, for *022.li*, the operation count was actually reduced. This anomaly is due to the unrolling done after loop peeling. Loop peeling affects the characteristics of loops that may be unrolled. If loop peeling makes

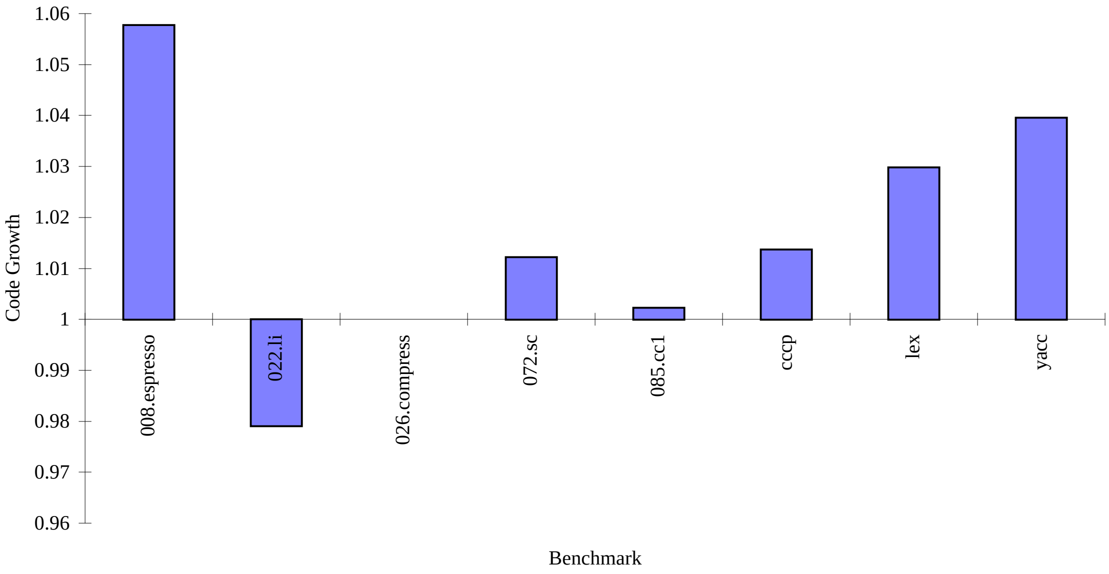


Figure 3.22: Loop peeling's effect on code size.

the loop more dense, the loop unrolling heuristic will unroll that loop less. Since the code growth that loop unrolling creates exceeds that of loop peeling, some loops may actually end up smaller than without loop peeling. Another factor that keeps code growth low is the *MAX_OPS_IN_PEELED_LOOP* parameter. Here it is set to 36 operations, meaning that no peeled loop exceeds 36 operations.

To better understand why benchmarks obtain different speedups due to loop peeling, two representative benchmarks are studied. The characteristics of *008.espresso* and *cccp* are shown in Figure 3.23. The benchmark *008.espresso* achieves a 38.6% speedup, while the benchmark *cccp* is sped up by less than 1%. This difference in performance is due to the fact that the peelable loops in *008.espresso* are the most frequently executed. The execution frequency of the peeled loops is indicated by the average iteration count

Characteristic	*008.espresso*	*cccp*
Benchmark Speedup	38.6%	0.4%
Average Total Iteration Count	69288	178
Average Size	11.27	5.68
Average Coverage	0.99	0.98
Average Peels	3.16	2.77

Figure 3.23: Characteristics of two benchmarks.

shown in Figure 3.23. The average iteration count of the peelable loops in *008.espresso* is hundreds of times larger than the average iteration count in *cccp*. The average size, coverage, and number of peels have a much smaller effect on the performance. Even if the loops in *cccp* could be peeled optimally, their net effect on performance is still limited by the average iteration count.

4. ADVANCED HYPERBLOCK OPTIMIZATIONS

This chapter presents a preliminary study of some advanced hyperblock optimization techniques. The techniques introduced here will be quantitatively assessed by implementing them in the IMPACT compiler. This chapter merely presents the motivation behind this future work.

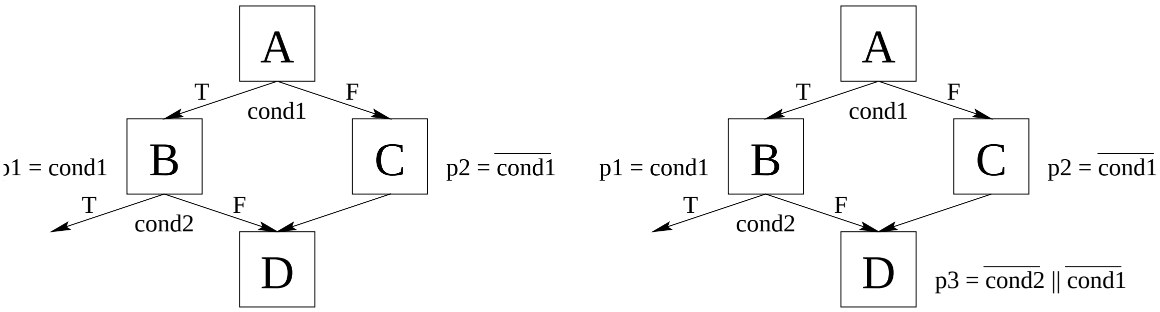
4.1 Fully Resolved Predicates

A superblock is a single-entry, multiple-exit region of code. Superblocks provide an efficient foundation for all phases of ILP compilation, including optimization, scheduling, and register allocation. They are formed by combining the most frequently taken path through a trace into one block. Hyperblocks are a generalized form of superblocks that allow multiple paths of execution through the use of predication.

By their nature, typical hyperblocks and superblocks include many infrequently taken exit branches. These infrequently taken branches impede code motion, increase the dependence height of the execution path, and increase the resource height of the block by consuming valuable branch resources. Note that elimination of these branches by merging in their taken paths results in a performance penalty since they are easily predicted and rarely taken. While removing these branches with if-conversion is unwise, we can still use predication to eliminate their negative effects.

Traditional if-conversion uses both predicates and branches to guard execution of operations in the resultant hyperblock. As a result, some control dependences remain in a completely predicated hyperblock. This type of if-conversion creates partially resolved predicates (PRPs). PRPs need control dependences created by the remaining branches to maintain code correctness. Alternatively, if-conversion can also be applied to create fully resolved predicates (FRPs). With FRPs, all instructions are guarded by predicates even if they were originally guarded by branches. The end result is a hyperblock with no control dependences remaining. In effect, all instructions can be scheduled without concern for the location of branches. Data dependences become the only concern during optimization and code scheduling.

Figure 4.1 illustrates the difference between a PRP predicated block and an FRP predicated block. The figure shows four blocks to be included in a hyperblock. In both the PRP and the FRP hyperblocks, both basic blocks, B and C, are predicated on the condition of the branch in block A. Block B contains another branch that, if taken, directs the flow of control outside of the hyperblock. In the PRP case, block D need not be predicated since its execution is guarded by the branch in B. In the FRP case, a predicate is created for block D regardless. The value of p3, the predicate for block D,



Partially Resolved Predicates

Fully Resolved Predicates

Figure 4.1: Example of partially and fully resolved predicates.

is $\overline{cond1}||\overline{cond2}$ since D will only be executed when control enters C or control enters B, and the branch in B is not taken.

Fully resolved predicates have interesting benefits when there is a need to speculate instructions above branches. Most models of control speculation incur a cost of some form or another to speculate an instruction. This cost is due to the fact that a speculated instruction should not produce any irreversible side effects when it would not have normally executed. For example, an instruction should not cause an exception when it was not supposed to execute. Since execution of all instructions is determined solely by predicates, instructions can be placed in any order as long as data dependence is respected. The need for a costly speculation model is reduced.

Moving instructions, which could potentially except or which could change memory state above branches, is important for obtaining high levels of instruction-level parallelism. While most speculation models adequately address the speculation of potentially

	CB 6:			Taken Frequency
1	r35 = MEM[r34]		branch r34 >= r37, CB 95	14
2	r34 = r34 + 1			
3			branch r35 == 10, CB 11	4035
4			branch r35 == 0, CB 11	0
5			branch r33 >= r57, CB 11	0
6	MEM[r33] = r35	r33 = r33 + 1	jump CB 6	101148

Figure 4.2: Original scheduled superblock code segment.

excepting instructions, speculating stores requires even more hardware or cannot be performed at all. FRP predicated hyperblocks are not subject to this restriction. Stores, like any other instruction, can be moved freely above branches.

No speculation model alone could handle the reordering of branches. Reordering of branches requires complicated compilation techniques and costly code duplication. Branches like all other instructions can be naturally handled by an FRP predicated hyperblock.

An example of FRP predication applied to a real code segment is shown in Figure 4.2. This code segment is from the function *execute* in the *grep* benchmark. This inner loop accounts for about 40% of the total execution of the program. A 3-issue, one branch-per-cycle machine with HP PA-7100 latencies is assumed for the schedule.

One thing to notice about this code segment is that all of the branches in this code segment are easily predicted. As indicated by the taken frequency column, the branches are almost never taken. The backedge jump, by necessity, is always taken when reached.

CB 6:

							Taken Frequency
1	r35 = MEM[r34]		$(p0_{ut}, p1_{uf})$ = (r34 >= r37)				
2	r34 = r34 + 1	(p1)			jump CB 95	(p0)	14
3	$(p2_{ut}, p3_{uf})$ = (r35 == 10)	(p1)					
4	$(p4_{ut}, p5_{uf})$ = (r35 == 0)	(p3)			jump CB 11	(p2)	4035
5	$(p6_{ut}, p7_{uf})$ = (r33 >= r57)	(p5)			jump CB 11	(p4)	0
6	MEM[r33] = r35	(p7)	r33 = r33 + 1	(p7)	jump CB 6	(p7)	101148
7					jump CB 11	(p6)	0

Figure 4.3: Scheduled FRP predicated hyperblock.

On average, six cycles are needed to complete one iteration of this loop. If the unconditional backedge could be promoted above one or more of the other branches, then the average iteration cycle count would be reduced as well.

FRP predication is applied to this code segment, which results in the schedule shown in Figure 4.3. The first thing to notice is the introduction of predicate defining instructions.[1] These instructions use the conditional expressions previously contained in the branch instructions of Figure 4.2 to compute and write values into predicate registers. These predicates are then used to guard the execution of instructions previously guarded by branches. Since a predicate must be defined at least one cycle before it is used, the entire schedule is increased by one cycle. To offset this performance penalty, the backedge is moved up one cycle, to its original position. The net performance change is 0.99, a slight loss. This loss is due to the first and second branches moving down one cycle.

If the backedge could be moved up more, the FRP predicated case would realize a performance win. The reason that the backedge cannot be moved up further is because

[1] The predicate defining instructions used here are based upon the HPL Playdoh architecture, which is defined in detail in [34]. UT writes the result value of the comparison. UF writes the complement of the comparison's result. Both UT and UF write FALSE if the input predicate is FALSE. AC only writes FALSE if both the input predicate and result of comparison are FALSE. Otherwise, AC does nothing.

of the dependence chain created by the predicate defining instructions. This dependence chain is highlighted. The value in p7 is needed by the backedge branch. To calculate p7, p5 must be calculated. In turn, to calculate p5, p3 must be computed. If p7 could be calculated more quickly, the backedge could be scheduled earlier. While there are many simple examples that demonstrate the benefit of FRPs directly, this example was chosen because it motives the discussion on height reduction contained in the next section.

4.2 Height Reduction

A fairly common technique to reduce dependence height is arithmetic height reduction. Arithmetic height reduction takes an imbalanced tree of computation and converts it to a minimum height tree. For example, $(((a * b) * c) * d)$ becomes $((a * b) * (c * d))$. Assuming single cycle latency for multiplication, this transformation reduces the dependence height from three to two cycles. A chain of predicate definitions can also be height reduced. However, since multiple OR and AND type predicate defining instructions with the same destination can be executed in a single cycle, the height of computation can be dramatically reduced.

Figure 4.4 shows the code segment after height reduction. Since it is desirable to move the backedge as early as possible, the compiler reduces the height of p7's computation. This height-reduced computation is shown shaded in grey. With the value of p7 calculated in cycle 3, the backedge is scheduled in cycle 4. The net performance gain is 46% for a single iteration. It is interesting to note that this performance gain is magnified by loop

CB 6:

							Taken Frequency
1	r35 = MEM[r34]		($p0_{un}$, $p1_{uc}$) = (r34 >= r37)		($p7_{uc}$) |= (r34 >= r37)		
2	r34 = r34 + 1	(p1)	($p7_{ac}$) |= (r33 >= r57)		jump CB 95	(p0)	14
3	($p2_{ut}$, $p3_{uf}$) = (r35 == 10)	(p1)	($p7_{ac}$) |= (r35 == 10)		($p7_{ac}$) |= (r35 == 0)		
4	MEM[r33] = r35	(p7)	r33 = r33 + 1	(p7)	jump CB 6	(p7)	101148
5	($p4_{ut}$, $p5_{uf}$) = (r35 == 0)	(p3)			jump CB 11	(p2)	4035
6	($p6_{ut}$) = (r33 >= r57)	(p5)			jump CB 11	(p4)	0
7					jump CB 11	(p6)	0

Figure 4.4: Scheduled FRP predicated hyperblock with height reduction.

unrolling in a wider machine. The original unpredicated code segment cannot be reduced to fewer then six cycles per iteration due to the limited branch resources. However, the height-reduced code ultimately results in fewer than four cycles per iteration.

Arithmetic height reduction is usually a win since it uses a similar amount of instruction resources but reduces the dependence height. The benefit of the height reduction technique shown here is not always guaranteed. This transformation and other height reducing transformations trade dependence height for operation count. A balance must be made to properly apply height reduction to obtain the best performance. The plan to address this problem is discussed later.

4.3 Node Splitting

Hyperblocks are formed by taking multiple paths through the control flow graph and combining them into a single entry block. Execution of these separate paths is then guarded by predicates. In general, these paths do not have the same dependence height or resource requirements. If two of these paths share a common dataflow merge point,

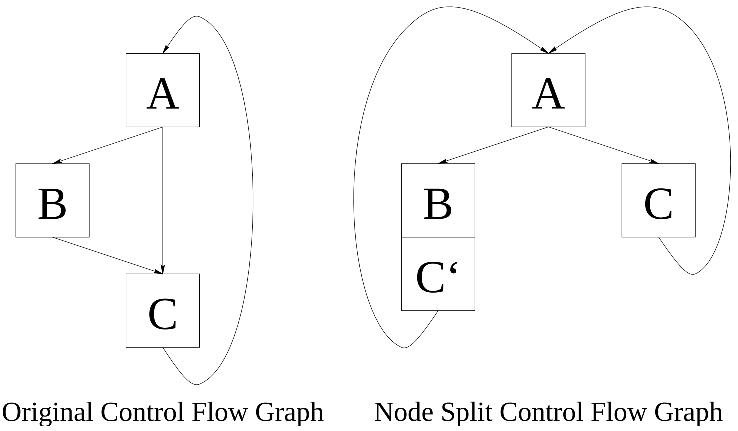


Figure 4.5: Node splitting concept.

then that merge point must be scheduled after the end of the longer of the two paths. In effect, the compiler is penalizing the shorter path for the sake of the longer path.

A typical situation is shown in the left side of Figure 4.5. There are two paths of execution through this loop: path ABC and path AC. Path ABC is necessarily longer than path AC. In the final scheduled code, block C cannot be scheduled before the end of block B, which leaves a gap between A and C that penalizes performance every time the program takes path AC.

We can remedy the situation by node splitting. Block C is duplicated as shown in the right side of Figure 4.5. This duplicate block is named C'. In the node split code, block C can now be scheduled immediately after block A.

Figures 4.6-4.8 show an example of node splitting from a function named *compress* in the benchmark *compress*. Note that in Figure 4.6 there are two virtual paths that exist in this predicated code segment. One path includes the instruction in cycle 3; the other does not. Notice that when this instruction is not executed, the instructions in cycles 4-7

1	r9 = r9 - r12	
2	$(p1_{uf})$ = (r9 < 0)	
3		r9 = r9 + r13 (p1)
4	r10 = r9 << 2	
5	r114 = MEM[r10]	
6		
7	branch (r14 <> r8) CB 38	

Figure 4.6: Node splitting example: original code segment.

CB 38:

1	r9 = r9 - r12	
2	$(p1_{uf}, p2_{ut})$ = (r9 < 0)	
3	r110 = r9 << 2 (p2)	r9 = r9 + r13 (p1)
4	r114 = MEM[r110] (p2)	r10 = r9 << 2 (p1)
5		r14 = MEM[r10] (p1)
6	branch (r114 <> r8) CB 38 (p2)	
7		branch (r14 <> r8) CB 38 (p1)

Figure 4.7: Node splitting example: after node splitting.

r1312 = r13 - r12

CB 38:

1		r9 = r9 - r12	r1009 = r9 + r1312
2	r110 = r9 << 2	$(p1_{uf}, p2_{ut})$ = (r9 < 0)	r10 = r1009 << 2
3	r114 = MEM[r110] (p2)	r9 = r1009 (p1)	r14 = MEM[r10] (p1)
4			r10 = r9 << 2 (p1)
5	branch (r114 <> r8) CB 38 (p2)		branch (r14 <> r8) CB 38 (p1)

Figure 4.8: Node splitting example: after node splitting and further optimization.

were delayed a cycle for no real reason. Figure 4.7 shows the code after node splitting. A copy of the shaded region in Figure 4.6 was made and started a cycle earlier. Notice that this saves one cycle every time p1 evaluates to FALSE. The resulting code has a very small performance gain. A code segment with more disparate path heights may seem to be a better example of node spitting. However, this code segment was chosen because it also illustrates another secondary benefit of node splitting.

When the original code segment is optimized, it remains relatively unchanged. However, when the node split code is optimized, a drastically different code segment results. This resulting code segment is shown in Figure 4.8. The reason so much more optimization is performed lies in the fact that for both paths the exact expression of R9 is known. In the left path of Figure 4.7, R9 is unchanged. In the right path, R9 is incremented by R13. This information is used by the compiler to perform expression optimization that was not possible without node splitting. The resulting code takes only five cycles per iteration, down from seven cycles.

4.4 Partial Reverse If-Conversion

Hyperblock formation is done by considering each path for inclusion in the hyperblock. If the anticipated benefit of including a path outweighs its potential harm, it is included. Unfortunately, this decision cannot possibly be based upon all of the pertinent information at hyperblock formation time. Optimizations in phases after hyperblock formation may change the resource utilization or dependence height of the resulting code.

If hyperblocks are aggressively formed by including extra paths, more optimizations may be possible. Additionally, the larger scheduling scope could expose more instruction-level parallelism. Conversely, the aggressively formed hyperblock might also result in a longer schedule length if extra optimization opportunities are unrealized. Too many paths can saturate the processor's available resources, or the dependence height of some paths may penalize other paths as was demonstrated in the node splitting example.

The solution to this problem is to form aggressive hyperblocks for better optimization potential yet have a way to minimize or eliminate any performance penalties that may result. A study of a method called partial reverse if-conversion (PRIC) is planned.

By giving the hyperblock formation heuristic the tendency to aggressively include paths, large performance wins result when optimization potential is realized. To account for the cases in which optimizations are not effective, partial reverse if-conversion excludes the portions of paths that are the cause of poor performance. Since the scheduler is the earliest phase of the compiler that knows exactly what the instruction schedule will look like, partial reverse if-conversion will most likely be guided by the scheduler. Optimization during scheduling, including PRIC, is discussed as part of Section 4.6.

The concept of partial reverse if-conversion is illustrated in Figure 4.9. The left-hand side of the figure shows a control flow graph before if-conversion. The current conservative IMPACT hyperblock formation algorithm will form a hyperblock with only path ABD. Path C will be excluded since it has a relatively low execution frequency and its height could penalize path ABD. An aggressive hyperblock formation algorithm would include

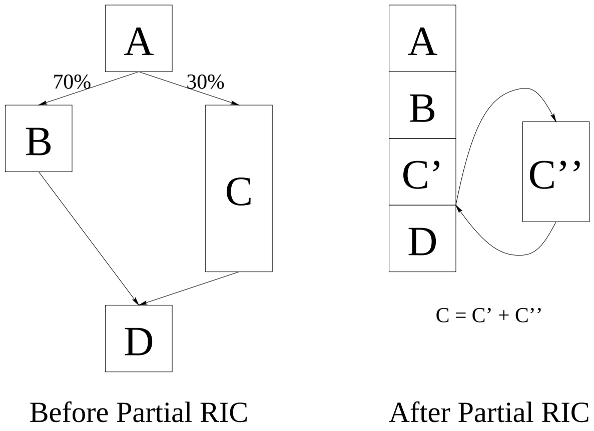


Figure 4.9: Partial reverse-if conversion concept.

paths ABD and ACD. There are two scenarios for what can happen in the following phases of the compiler. In the first scenario, both paths are optimized, and the resulting schedule is a win for both paths. In the second scenario, the schedule of ACD remains relatively large compared to ABD. In this case, path ABD is unnecessarily penalized. To remedy this situation, PRIC is applied.

The right-hand side of Figure 4.9 shows the code after PRIC. During scheduling, block A and the top portion of blocks B and C are scheduled in priority order. As the final instruction from block B is placed, the scheduler notices that the remaining height of C is large. The PRIC heuristic decides to insert a branch that jumps to the portion of C remaining, C”, which is located in another part of the program. Block D may then be tail duplicated, or block C” can conclude with a jump back into the original hyperblock.

An inspection of the schedules for the IMPACT compiler suggests that PRIC should be very effective. When some selected regions are formed into aggressive hyperblocks, the resulting performance of the benchmark is markedly improved. In addition, some regions formed with the current IMPACT compiler have poor performance due to aggressive formation. PRIC would rectify this situation.

4.5 Profile Independent Hyperblock Selection

The current IMPACT compiler uses profiling to determine the behavior of the code it compiles. Frequently executed regions revealed by profiling are given priority in the optimizations performed. In general, profiles based on one input set are useful in predicting the program's behavior with new input sets. In a research compiler such as IMPACT, profiling allows the researcher to effectively test a new optimization. If a heuristic is used to determine the behavior of a program, then the test of the new optimization may be affected by the effectiveness of the profile heuristic. In effect, profiling creates one less heuristic dimension. The IMPACT compiler framework allows the use of a static profiler when available.

Despite the advantages of dynamic profiling, applications exist for compilers where dynamic profiling is inappropriate. In fact, most commercial compilers do not support dynamic profiling at all. For this reason, it is important to study ways to make IMPACT's predicate optimizations profile independent.

There are many ways to avoid the use of dynamic profile information. Static profiling, as applied to superblock formation, has been studied by Hank [37]. This paper shows that static profiling is effective yet not as effective as dynamic profiling. Hyperblock formation has some interesting differences to superblock formation that could make it an interesting application of static profiling. Superblock formation requires that the bias of a branch be determined by the static profiler. However, hyperblock formation has the option of including multiple paths if a bias cannot be determined. This freedom reduces the need for a highly accurate static profile estimate and could make static profiling more competitive to dynamic profiling.

Execution profiling is not the only valuable measure in making optimization decisions. Dependence height and resource utilization can also be used to enhance the performance of programs. For example, comparing the heights of paths considered for inclusion in a hyperblock is important. Guiding optimizations by dependence height and resource utilization requires information about the machine being compiled for. Since good compilers are portable, it is wise to use a universal machine description language to detail the target machine to the compiler. The next section details future research into driving optimization with machine descriptions.

4.6 Optimization at Schedule Time

While a machine description-driven optimization can do a satisfactory job of estimating the final schedule length of a code segment, the final schedule length cannot be known before scheduling is performed. Since the scheduler knows exactly what resources remain available, it could also know which optimizations would benefit the final schedule. This knowledge makes the scheduler ideal for directing optimizations.

The problem with optimization at schedule time is that code already scheduled may have to change when an optimization is performed. In this environment, the scheduler must be able to unschedule operations and continue scheduling after a transformation without having to restart from the beginning. It also requires that alternatively scheduling and optimizing converges on a solution. Satisfying these requirements is the major challenge in effectively guiding optimizations at schedule time.

While optimization during scheduling is more accurate for some optimizations, it is essential for others. Partial reverse if-conversion is basically not possible without guidance from the scheduler. During partial reverse if-conversion, the location at which one path extends the length of another path must be known. This location is determined at schedule time. Therefore, PRIC must be guided by the scheduler. It is for this reason that a scheduler framework that allows optimization at schedule time must be developed.

5. CONCLUSIONS

This thesis has described advanced hyperblock optimizations. Loop peeling was chosen as the representative optimization for detailed study. This detailed study showed how a concept such as loop peeling could be applied in practice. It demonstrated how an optimization applied alone may decrease performance yet expose opportunities that eventually lead to overall code speedup. The other optimizations presented in this thesis will be analyzed through further study and implementation in a similar manner. The end result will be an advanced second-generation compiler for machines with predication support.

This work, combined with the work of others in the field, shows that predication is an extremely valuable tool in extracting instruction-level parallelism from programs.

REFERENCES

[1] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 272–282.

[2] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 290–302.

[3] M. A. Schuette and J. P. Shen, "An instruction-level performance analysis of the Multiflow TRACE 14/300," in *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture*, November 1991, pp. 2–11.

[4] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 386–395.

[5] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, no. 1, pp. 12–35, January 1989.

[6] R. A. Towle, "Control and data dependence for program transformations," Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1976.

[7] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, January 1983, pp. 177–189.

[8] J. C. Park and M. S. Schlansker, "On predicated execution," Hewlett Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-91-58, May 1991.

[9] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992, pp. 45–54.

[10] S. A. Mahlke, R. E. Hank, J. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995, pp. 138–150.

[11] D. C. Lin, "Compiler support for predicated execution in superscalar processors," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.

[12] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of conditional branches," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[13] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 40–51.

[14] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau, "Reverse if-conversion," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993, pp. 290–299.

[15] G. E. Haab, "Data dependence analysis for Fortran programs in the IMPACT compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[16] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[17] K. Subramanian, "Loop transformations for parallel compilers," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[18] S. Anik, "Architectural and software support for executing numerical applications on high performance computers," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[19] Y. Yamada, "Data relocation and prefetching for programs with large data sets," Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[20] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th International Symposium on Computer Architecture*, May 1989, pp. 242–251.

[21] P. P. Chang, "Compiler support for multiple instruction issue architectures," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[22] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Profile-guided automatic inline expansion for C programs," *Software Practice and Experience*, vol. 22, no. 5, pp. 349–370, May 1992.

[23] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[24] R. A. Bringmann, "Compiler-controlled speculation," Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[25] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 353–370, March 1995.

[26] N. J. Warter, "Modulo scheduling with isomorphic control transformations," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[27] W. Y. Chen, "Data preload for superscalar and VLIW processors," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[28] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1994, pp. 183–193.

[29] R. E. Hank, "Machine independent register allocation for the IMPACT-I C compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[30] J. C. Gyllenhaal, "A machine description language for compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[31] R. A. Bringmann, "Template for code generation development using the IMPACT-I C compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1992.

[32] W. Y. Chen, "An optimizing compiler code generator: A platform for RISC performance analysis," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[33] R. G. Ouellette, "Compiler support for SPARC architecture processors," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[34] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Hewlett-Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-93-80, February 1994.

[35] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Supercomputing*, November 1990.

[36] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: An introduction and motivation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 158–168.

[37] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu, "Superblock formation using static program analysis," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993, pp. 247–255.